Distributed Programming with Erlang - a crash course

Cons T Åhs
cons@klarna.com
@lisztspace

Cons T Åhs

- Senior developer/architect at Klarna since feb 2011
  - Architecture, development and Code Quality
  - Increasing competence of developers
- Previously
  - consultant; online poker, low level networking, medical imaging, graphics, finance, musical notation, compilers, real time video decoding, teaching..
  - lecturer at Uppsala University, research & teaching; foundations, algorithms, functions, relations, objects, compilers, pragmatics, theory, theorem proving, formal program correctness..

Klarna

Klarna - the business

- Make shopping on the net simpler, safer, more fun.
- Pay by invoice after the goods delivered
- Two levels of customers
- End consumers
- estores
- Customer checks out at estore
- klarna identifies customer and investigates credit
- estore sends goods and invoice
- klarna pays estore (klarna takes the risk)
- customer pays klarna

Klarna - the facts

- Founded in 2005
- Revenue doubled every year from start
- Sweden, Norway, Denmark, Finland, Germany, Netherlands
- Over 800 employees
- Over 15K estores
- Currently 2-3M transactions/month
- Available 24/7 - no downtime
- software upgrades with no downtime
- hardware upgrades and relocation with no downtime

Erlang - The Language

- Conceived at Ericsson
- Buzzword compliancy
  - Functional - no side effects
  - Robust - built for fault tolerance and high availability
  - Runs in a virtual machine (VM) called beam
  - Extremely lightweight processes - from 309 words
  - Easy to distribute among cores, VMs and machines
  - No shared memory between processes
  - Processes communicate asynchronously through mail boxes
  - OTP - Open Telecom Platform

Klarna

A Functional Language

- Dynamically typed functional language
- No side effects; variables are bound once and the value can not be changed
  - trying to reassign a variable will crash the program
- Every expression computes a value
- Pattern matching provides parallel binding and compact programs (mixed blessing - beware!)
- Looks very much like Prolog
  - A function is determined by both name and arity
  - functions are divided in clauses
  - function bodies are sequences of expressions
- The power of higher order functions and closures

Klarna

Basic Workings

- The file `example.erl` holds module `example`
- The exported functions constitutes the interface of the module
- Access exported functions `module:fun(<args>)`
- Erlang is started with `erl` presenting you with a basic REPL (read-eval-print loop)
  - enter expressions and see value
- Use `c/1` to compile a file

Klarna

Compute length of list

```
-module(ex1).

-export([ rlen/1
        , tlen/1
        ]).

%% Ordinary recursive definition
rlen([])      -> 0;
rlen([_| L]) -> 1 + rlen(L).

%% Tail recursive definition
tlen(L) ->
  tlen(L, 0).

%% Tail recursive help function
tlen([], N)      -> N;
tlen([_| L], N) -> tlen(L, N+1).
```

Klarna

Data representation

- Data is built from numbers, atoms, tuples and lists
    - `11, 42, 4711, 3.141692657`
    - `foo, klarna, invoice, last_name, false`
    - `{foo, 12}`
    - `{ray, {vec, 0.0, 1.0, 1.2}, {vec, 1, 1, 1}}`
    - `[foo, bar, baz]`
    - `[{object, 12}, wall, {true, 42}]`
- Strings are just lists of characters (!)
- There is some support for abstraction in the form of records
- Also, opaque data such as pids, binaries, refs

Klarna

Insert into ordered tree

```
-module(ex2).

-export([cinsert/2]).

-record(tree, {info, left=empty, right=empty}).

cinsert(E, empty) -> #tree{info = E};
cinsert(E, T = #tree{info = E}) -> T;
cinsert(E, T = #tree{info = I}) when E < I ->
  T#tree{left = cinsert(E, T#tree.left)};
cinsert(E, T = #tree{info = I}) when E > I ->
  T#tree{right = cinsert(E, T#tree.right)}.
```

Klarna

Conditional computation

- Pattern matching in clauses (possibly using guards)
  - compact code - might be good
  - explicit representation - definitely bad
- case expression
  - inline pattern matching on result of expression
- if expression
  - prime example of lack of insight of language design

Klarna

Abstract insert

```
-module(ex3).

-export([ empty_tree/0, insert/2]).

-record(tree, {info, left=empty, right=empty}).

empty_tree()               -> empty.
tree_info(#tree{info = I}) -> I.
tree_left(#tree{left = L}) -> L.
tree_right(#tree{left = R}) -> R.

is_empty_tree(empty)    -> true;
is_empty_tree(#tree{}) -> false.

mk_node(E)                -> #tree{info = E}.
mk_tree(E, Left, Right) -> #tree{info = E, left = Left, right = Right}.

insert(E, Tree) ->
  case is_empty_tree(Tree) of
    true  -> mk_node(E);
    false ->
      I = tree_info(Tree),
      if E == I -> Tree;
         E < I ->
          mk_tree(I, insert(E, tree_left(Tree)), tree_right(Tree));
         true ->
          mk_tree(I, tree_left(Tree), insert(E, tree_right(Tree)))
      end
  end.
```

Similar syntax, different meaning

Are these all the same?

No.
The types are different

```
is_empty_tree(empty)    -> true;
is_empty_tree(#tree{}) -> false.
```

```
empty | #tree -> true | false
```

```
is_empty_tree(empty) -> true;
is_empty_tree(_)     -> false.
```

```
any() -> true | false
```

```
is_empty_tree(Tree) -> Tree == empty.
```

```
any() -> true | false
```

```
is_empty_tree(Tree) -> Tree = empty.
```

```
empty -> empty
```

The last two shows the difference between binding and matching

Klarna

Higher order functions

- Functions are first class citizens
  - a variable can be bound to a function
  - a function can be the result of a computation
  - a function can be passed as an argument

```
-module(ex4).

-export([ sorttuples/1
        ]).

sorttuples(Tuples) ->
  Num = fun({_, N}) -> N end,
  Cmp = fun(T1, T2) -> Num(T1) < Num(T2) end,
  lists:sort(Cmp, Tuples).
```

Klarna

Concurrent and distributed programming

- With concurrent programming troubles form when you have a shared <u>and</u> mutable state.
- Problem typically solved by using synchronisation with locks
    - Complicated - you have to know when to lock
    - Can lead to more problems - performance degradation
    - Cooperative model - all parts of the program must agree
- Take away one and your on safe ground.
- Erlang takes away both!

No shared state, no mutable state

- Each process has a state of its own, or rather a sequence of states; possibly a new state after receiving a message
- Each process has a private heap
- Each process has a message queue (the implementation handles these)
- Processes can not share state, even when they live in the same VM.
  - All communication must be done with messages.
  - messages are copied between processes

Klarna

No shared state

- Why?
    - Background (telecom switches) with a large number of small and short lived processes
    - When a process dies there is no risk reclaiming the whole process
    - No other process can access the memory it used
    - Nothing happens if you send a message to a dead pid
    - The dead process can not reference the memory of another process
    - Leads to robustness

Klarna

Keeping state in a process

- Real world computations need state
- State is encoded in a process that reacts to messages
  - init state
  - wait for message
  - compute new state from message and existing state
  - loop

```
start() -> actor(init_state()).

actor(State) ->
  actor(process_message(get_msg(), State)).
```

- start the actor and send messages to it

Klarna

Managing processes

- Three basic primitives are used to handle processes
- Create process - returns pid (process id)

```
spawn(Function)
```

- Send a message - returns Msg (without waiting)

```
Pid ! Msg
```

- Receive a message - returns value of chosen expression

```
receive
  Pattern1 -> Expr1;
  Pattern2 -> Expr2;
  ...
end
```

Klarna

Efficient computation through memoisation

- Consider a computationally intensive function
  - Fibonacci, Ackermann, ..
- Instead of computing the value each time, one can remember the values and serve them when a new request comes
  - If we know the value, return it
  - Otherwise, compute it, remember it, return it
- It's actually a cache!
- The cache (a mapping from argument(s) to value) is encoded in the state of a process

Klarna

Efficient computation through memoisation

```
-module(ex5).

-export([ fib/1, fibfun/0]).

fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).

fibfun() ->
  Cache = dict:new(),
  Pid = spawn(fun() -> loop(Cache) end),
  fun(N) ->
      Pid ! {self(), N},
      receive
        V -> V
      end
  end.

loop(Cache) ->
  receive
    {Pid, N} ->
      case dict:find(N, Cache) of
        {ok, Value} ->
          NewCache = Cache;
        error ->
          Value = fib(N),
          NewCache = dict:store(N, Value, Cache)
      end,
      Pid ! Value,
      loop(NewCache)
  end.
```

Distribution made easy

- Distribute work load among a number of workers
- Input
  - the work to be done, a queue of tasks
  - the workers that performs the work (pids)
- What is specific for each problem?
  - How to get a chunk of work from the queue
  - How to combine results from a single worker with the result from the others

Klarna

Distribution made easy

- We're done when the queue is empty and we have no active workers.
- We wait for a worker to return a result when the queue is empty or we have no passive workers
- We activate a worker when the queue is non empty and we have passive workers.
- Initial state is a queue of work, no active workers and a collection of passive workers.

Klarna

Distribution made easy

```
sequential(L) -> lists:filter(fun is_prime/1, L).

process_work([], [], _, State) -> State;
process_work(Work, Active, Passive, State)
  when Work =:= []; Passive =:= [] ->
  receive {Worker, M} ->
      process_work(Work, lists:delete(Worker, Active),
                   [Worker | Passive], add_result(State, M))
  end;
process_work(Work, Active, [Worker | Passive], State) ->
  {Chunk, Rest} = get_chunk(State, Work),
  Worker ! {self(), Chunk},
  process_work(Rest, [Worker | Active], Passive, State).

worker() ->
  receive {Pid, Work} ->
      Pid ! {self(), sequential(Work)},
      worker()
  end.
```

Klarna

More about Erlang

- Covered the basics of Erlang and distributed and concurrent programming
- OTP, Supervisors, behaviours, gen_server, rebar, eunit, proper, dialyzer, standard libraries, persistence in various forms, bit syntax, code loading, actual side effects ..
- Good book
  - Erlang and OTP in Action by Martin Logan, Eric Meritt, Richard Carlsson.

Klarna

More about Klarna

- http://engineering.klarna.com/
- signmeup@klarna.com